MultiMarkdown v6 Development Notes

Fletcher T. Penney

2017-03-15

Introduction

This document includes some notes on the development of Multi-Markdown (MMD) v6. Most of it will be interesting only to other developers or those needing to choose the absolute "best" Markdown (MD) implementation for their needs – it is not required reading to understand how the software works.

Why a New Version?

MultiMarkdown version 5 was released in November of 2015, but the codebase was essentially the same as that of v4 – and that was released in beta in April of 2013. A few key things prompted work on a new version:

- Accuracy – MMD v4 and v5 were the most accurate versions yet, and a lot of effort went into finding and resolving various edge cases. However, it began to feel like a game of whack-a-mole where new bugs would creep in every time I fixed an old one. The PEG began to feel rather convoluted in spots, even though it did allow for a precise (if not always accurate) specification of the grammar.

- Performance – "Back in the day" peg-markdown[1] was one of the fastest Markdown parsers around. MMD v3 was based on peg-markdown, and would leap- frog with it in terms of performance. Then CommonMark[2] was released, which was a bit faster. Then a couple of years went by and CommonMark became *much* faster – in one of my test suites, MMD v 5.4.0 takes about 25 times longer to process a long document than CommonMark 0.27.0.

  In the spring of 2016, I decided I wanted to rewrite MultiMark-down from scratch, building the parser myself rather than relying on a pre-rolled solution. (I had been using greg[3] to compile the PEG into parser code. It worked well overall, but lacked some features I needed, requiring a lot of workarounds.)

First Attempt

My first attempt started by hand-crafting a parser that scanned through the document a line at a time, deciding what to do with each

[1] https://github.com/jgm/peg-markdown

[2] http://commonmark.org/

[3] https://github.com/ooc-lang/greg

line as it found them. I used regex parsers made with re2c[4] to help classify each line, and then a separate parser layer to process groups of lines into blocks. Initially this approach worked well, and was really efficient. But I quickly began to code my way into a dead-end – the strategy was not elegant enough to handle things like nested lists, etc.

One thing that did turn out well from the first attempt, however, was an approach for handling <emph> and <strong> parsing. I've learned over the years that this can be one of the hardest parts of coding accurately for Markdown. There are many examples that are obvious to a person, but difficult to properly "explain" how to parse to a computer.

No solution is perfect, but I developed an approach that seems to accurately handle a wide range of situations without a great deal of complexity:

1. Scan the documents for asterisks (∗). Each one will be handled one at a time.

2. Unlike brackets ([and]), an asterisk is "ambidextrous", in that it may be able to open a matched pair of asterisks, close a pair, or both. For example, in `foo *bar* foo`:

   (a) The first asterisk can open a pair, but not close one.

   (b) The second asterisk can close a pair, but not open one.

3. So, once the asterisks have been identified, each has to be examined to determine whether it can open/close/both. The algorithm is not that complex, but I'll describe it in general terms. Check the code for more specifics. This approach seems to work, but might still need some slight tweaking. In the future, I'll codify this better in language rather than just in code.

   (a) If there is whitespace to the left of an asterisk, it can't close.

   (b) If there is whitespace or punctuation to the right it can't open.

   (c) "Runs" of asterisks, e.g. `**bar` are treated as a unit in terms of looking left/right.

   (d) Asterisks inside a word are a bit trickier – we look at the number of asterisks before the word, the number in the current run, and the number of asterisks after the word to determine which combinations, if any, are permitted.

4. Once all asterisks have been tagged as able to open/close/both, we proceed through them in order:

[4] http://re2c.org/index.html

(a) When we encounter a tag that can close, we look to see if there is a previous opener that has not been paired off. If so, pair the two and remove the opener from the list of available asterisks.

(b) When we encounter an opener, add it to the stack of available openers.

(c) When encounter an asterisk that can do both, see if it can close an existing opener. If not, then add it to the stack.

5. After all tokens in the block have been paired, then we look for nesting pairs of asterisks in order to create <emph> and <strong> sets. For example, assume we have six asterisks wrapped around a word, three in front, and three after. The asterisks are indicated with numbers: `123foo456`. We proceed in the following manner:

   (a) Based on the pairing algorithm above, these asterisks would be paired as follows, with matching asterisks sharing numbers – `123foo321`.

   (b) Moving forwards, we come to asterisk "1". It is followed by an asterisk, so we check to see if they should be grouped as a <strong>. Since the "1" asterisks are wrapped immediately outside the "2" asterisks, they are joined together. More than two pairs can't be joined, so we now get the following – `112foo211`, where the "11" represents the opening and closing of a <strong>, and the "2" represents a <emph>.

6. When matching a pair, any unclosed openers that are on the stack are removed, preventing pairs from "crossing" or "intersecting". Pairs can wrap around each other, e.g. `[(foo)]`, but not intersect like `[(foo])`. In the second case, the brackets would close, removing the `(` from the stack.

7. This same approach is used in all tokens that are matched in pairs– `[foo]`, `(foo)`, `_foo_`, etc. There's slightly more to it, but once you figure out how to assign opening/closing ability, the rest is easy. By using a stack to track available openers, it can be performed efficiently.

In my testing, this approach has worked quite well. It handles all the basic scenarios I've thrown at it, and all of the "basic" and "devious" edge cases I have thought of (some of these don't necessarily have a "right" answer – but v6 gives consistency answers that seem as reasonable as any others to me). There are also three more edge cases I've come up can still stump it, and ironically they are handled correctly by most implementations. They just don't follow the rules above. I'll continue to work on this.

In the end, I scrapped this effort, but kept the lessons learned in the token pairing algorithm.

Second Attempt

I tried again this past Fall. This time, I approached the problem with lots of reading. *Lots and lots* of reading – tons of websites, computer science journal articles, PhD theses, etc. Learned a lot about lexers, and a lot about parsers, including hand-crafting vs using parser generators. In brief:

1. I learned about the Aho–Corasick algorithm[5], which is a great way to efficiently search a string for multiple target strings at once. I used this to create a custom lexer to identify tokens in a MultiMarkdown text document (e.g. *, [, {++, etc.). I learned a lot, and had a good time working out the implementation. This code efficiently allowed me to break a string of text into the tokens that mattered for Markdown parsing.

   [5] https://en.wikipedia.org/wiki/Aho-Corasick_algorithm

2. However, in a few instances I really needed some features of regular expressions to simplify more complex structures. After a quick bit of testing, using re2c to create a tokenizer was just as efficient, and allowed me to incorporate some regex functionality that simplified later parsing. I'll keep the Aho-Corasick stuff around, and will probably experiment more with it later. But I didn't need it for MMD now. `lexer.re` contains the source for the tokenizer.

I looked long and hard for a way to simplify the parsing algorithm to try and "touch" each token only once. Ideally, the program could step through each token, and decide when to create a new block, when to pair things together, etc. But I'm not convinced it's possible. Since Markdown's grammar varies based on context, it seems to work best when handled in distinct phases:

1. Tokenize the string to identify key sections of text. This includes line breaks, allowing the text to be examined one line at time.

2. Join series of lines together into blocks, such as paragraphs, code blocks, lists, etc.

3. The tokens inside each block can then be paired together to create more complex syntax such as links, strong, emphasis, etc.

To handle the block parsing, I started off using the Aho-Corasick code to handle my first attempt. I had actually implemented some basic regex functionality, and used that to group lines together to create blocks. But this quickly fell apart in the face of more complex

structures such as recursive lists. After a lot of searching, and *tons* more reading, I ultimately decided to use a parser generator to handle the task of group lines into blocks. `parser.y` has the source for this, and it is processed by the lemon[6] parser generator to create the actual code.

I chose to do this because hand-crafting the block parser would be complex. The end result would likely be difficult to read and understand, which would make it difficult to update later on. Using the parser generator allows me to write things out in a way that can more easily be understood by a person. In all likelihood, the performance is probably as good as anything I could do anyway, if not better.

Because lemon is a LALR(1) parser, it does require a bit of thinking ahead about how to create the grammar used. But so far, it has been able to handle everything I have thrown at it.

Optimization

One of my goals for MMD 6 was performance. So I've paid attention to speed along the way, and have tried to use a few tricks to keep things fast. Here are some things I've learned along the way. In no particular order:

Memory Allocation

When parsing a long document, a *lot* of token structures are created. Each one requires a small bit of memory to be allocated. In aggregate, that time added up and slowed down performance.

After reading for a bit, I ended up coming up with an approach that uses larger chunks of memory. I allocate pools of of memory in large slabs for smaller "objects"". For example, I allocate memory for 1024 tokens at a single time, and then dole that memory out as needed. When the slab is empty, a new one is allocated. This dramatically improved performance.

When pairing tokens, I created a new stack for each block. I realized that an empty stack didn't have any "leftover" cruft to interfere with re-use, so I just used one for the entire document. Again a sizeable improvement in performance from only allocating one object instead of many. When recursing to a deeper level, the stack just gets deeper, but earlier levels aren't modified.

Speaking of tokens, I realized that the average document contains a lot of single spaces (there's one between every two words I have written, for example.) The vast majority of the time, these single spaces have no effect on the output of Markdown documents. I

changed my whitespace token search to only flag runs of 2 or more spaces, dramatically reducing the number of tokens. This gives the benefit of needing fewer memory allocations, and also reduces the number of tokens that need to be processed later on. The only down-side is remember to check for a single space character in a few instances where it matters.

Proper input buffering

When I first began last spring, I was amazed to see how much time was being spent by MultiMarkdown simply reading the input file. Then I discovered it was because I was reading it one character at a time. I switched to using a buffered read approach and the time to read the file went to almost nothing. I experimented with different buffer sizes, but they did not seem to make a measurable difference.

Output Buffering

I experimented with different approaches to creating the output after parsing. I tried printing directly to `stdout`, and even played with different buffering settings. None of those seemed to work well, and all were slower than using the `d_string` approach (formerly call `GString` in MMD 5).

Fast Searches

After getting basic Markdown functionality complete, I discovered during testing that the time required to parse a document grew exponentially as the document grew longer. Performance was on par with CommonMark for shorter documents, but fell increasingly behind in larger tests. Time profiling found that the culprit was searching for link definitions when they didn't exist. My first approach was to keep a stack of used link definitions, and to iterate through them when necessary. In long documents, this performs very poorly. More research and I ended up using uthash[7]. This allows me to search for a link (or footnote, etc.) by "name" rather than searching through an array. This allowed me to get MMD's performance back to O(n), taking roughly twice as much time to process a document that is twice as long.

[7] http://troydhanson.github.io/uthash/

Efficient Utility Functions

It is frequently necessary when parsing Markdown to check what sort of character we are dealing with at a certain position – a letter, whitespace, punctuation, etc. I created a lookup table for this via

`char_lookup.c` and hard-coded it in `char.c`. These routines allow me to quickly, and consistently, classify any byte within a document. This saved a lot of programming time, and saved time tracking down bugs from handling things slightly differently under different circumstances. I also suspect it improved performance, but don't have the data to back it up.

Testing While Writing

I developed several chunks of code in parallel while creating MMD 6. The vast majority of it was developed largely in a test-driven development[8] approach. The other code was largely created with extensive unit testing to accomplish this.

MMD isn't particularly amenable to this approach at the small level, but instead I relied more on integration testing with an ever-growing collection of text files and the corresponding HTML files in the MMD 6 test suite. This allowed me to ensure new features work properly and that old features aren't broken. At this time, there are 29 text files in the test suite, and many more to come.

Other Lessons

Some things that didn't do me any good. . . .

I considered differences between using `malloc` and `calloc` when initializing tokens. The time saved by using `malloc` was basically exactly offset by the initial time required to initialize the token to default null values as compared to using `calloc`. When trying `calloc` failed to help me out (thinking that clearing a single slab in the object pool would be faster), I stuck with `malloc` as it makes more sense to me in my workflow.

I read a bit about struct padding[9] and reordered some of my structs. It was until later that I discovered the `-Wpadded` option, and it's not clear whether my changes modified anything. Since the structs were being padded automatically, there was no noticeable performance change, and I didn't have the tools to measure whether I could have improved memory usage at all. Not sure this would be worth the effort – much lower hanging fruit available.

Performance

Basic tests show that currently MMD 6 takes about 20–25% longer the CommonMark 0.27.0 to process long files (e.g. 0.2 MB). However, it is around 5% *faster* than CommonMark when parsing a shorter file (27 kB) (measured by parsing the same file 200 times over). This

[8] https://en.wikipedia.org/wiki/Test-driven_development

[9] http://www.catb.org/esr/structure-packing/

test suite is performed by using the Markdown [syntax page], modified to avoid the use of the Setext header at the top. The longer files tested are created by copying the same syntax page onto itself, thereby doubling the length of the file with each iteration.

The largest file I test is approximately 108 MB (4096 copies of the syntax page). On my machine (2012 Mac mini with 2.3 GHz Intel Core i7, 16 GB RAM), it takes approximately 4.4 seconds to parse with MMD 6 and 3.7 seconds with CommonMark. MMD 6 processes approximately 25 MB/s on this test file. CommonMark 0.27.0 gets about 29 MB/s on the same machine.

There are some slight variations with the smaller test files (8–32 copies), but overall the performance of both programs (MMD 6 and CommonMark) are roughly linear as the test file gets bigger (double the file size and it takes twice as long to parse, aka O(n)).

Out of curiosity, I ran the same tests on the original Markdown.pl by Gruber (v 1.0.2b8). It took approximately 178 seconds to parse 128 copies of the file (3.4 MB) and was demonstrating quadratic performance characteristics (double the file size and it takes 2^2 or 4 times longer to process, aka O(n^2)). I didn't bother running it on larger versions of the test file. For comparison, MMD 6 can process 128 copies in approximately 140 msec.

Of note, the throughput speed drops when testing more complicated files containing more advanced MultiMarkdown features, though it still seems to maintain linear performance characteristics. A second test file is created by concatenating all of the test suite files (including the Markdown syntax file). In this case, MMD gets about 13 MB/s. CommonMark doesn't support these additional features, so testing it with that file is not relevant. I will work to see whether there are certain features in particular that are more challenging and see whether they can be reworked to improve performance.

As above, I have done some high level optimization of the parse strategy, but I'm sure there's still a lot of room for further improvement to be made. Suggestions welcome!

Testing

Test Suite

The development of MMD v6 was heavily, but not absolutely, influenced by the philosophy of test-driven development. While coding, I made use of test suites to verify successful implementation of new features, to avoid regression problems when adding new features, and to identify known edge cases in need of proper handling.

The test suite (located in `tests/MMD6Tests`) is a "living" collection

of documents that will continue to be updated as new bugs and edge cases are identified. This helps make proper integration testing of the entire application with every release.

Fuzz Testing

I was not familiar with the concept of subsection (https://en.wikipedia.org/wiki/Fuzzing) until a user mentioned something about it to me a year or two ago. I had never used it before, but it seemed like a good idea. I implement it in two ways.

The first is that I created a simplified version of the line parser that simply accepts various combinations of line type identifiers to see if they would successfully parse. The line parser is responsible for taking a series of line types (e.g. plain text, indented line, etc.) and determining what sort of block they should become. The file `test/parser_text.y` is run through the `lemon` program, compiled (with or without the `-DNDEBUG` flag) and then run. It sequentially throws every combination of line types at the simplified line parser to make sure that it doesn't choke. When I first did this, I found several combinations of lines that did not pass.

NOTE: This does not verify accurate parsing, simply that the parser does not crash by an unacceptable combination of lines.

The second form of fuzz testing I have started using more recently. This is using the American fuzzy lop[10] program to try to find text input that crashes MMD. This works by taking sample input (e.g. files from the test suite), modifying them slightly, and trying the modified versions. Do this over and over and over, and some interesting edge cases are sometimes identified. I have found some interesting edge cases this way. Definitely a very useful tool!

[10] http://lcamtuf.coredump.cx/afl/

Unit Testing

Some of the original development was done with unit testing in some other tools I developed. This code formed the basis of a few parts of MMD. Otherwise, it was hard to see how to really create very good unit tests for the development of MMD. So there is really not much unit testing built into the code or used during the development.

Dependencies/Libraries

MMD v6 has no external dependencies when compiling, aside from the standard libraries for C development.

MMD can be compiled without any other tools beside the build system (cmake).

If you want to edit the block parser, you need to modify the `parser.y` file and process that using lemon[11] in order to update the `parser.c` file. The lemon parser source is actually included in MMD and needs to be compiled to be used.

[11] http://www.hwaci.com/sw/lemon/

If you want to update the lexer or scanner utility functions, then you can modify `lexer.re` or `scanners.re`. These need to be processed using re2c[12], which has to be installed separately.

[12] http://re2c.org/

MMD v6 makes use of several other projects to improve performance and ease of use:

- uthash[13] – provides support for hashes to quickly locate specific "objects" based on text keys. These are used for matching footnotes, links, images and the like when created with the reference syntax. Much faster than searching through each one sequentially.

  [13] https://troydhanson.github.io/uthash/

- miniz[14] – provides zip archive support to enable creation of EPUB 3 files, which are zip files with a specific file structure inside them.

  [14] https://github.com/richgel999/miniz

- argtable3[15] – provides more advanced command-line argument processing in a more cross-platform approach than that used by MMD v5.

  [15] https://github.com/argtable/argtable3

Changelog

- 2017–03–19 – v 6.0.0-rc2:

  - ADDED: Add debugging info
  - ADDED: Add image assets when creating EPUB
  - ADDED: Add mmd2epub convenience script
  - ADDED: Add support for random footnote numbers
  - ADDED: Include tables as targets for cross–reference links
  - ADDED: Store css metadata as asset
  - CHANGED: Table captions now bottom aligned in HTML
  - CHANGED: Update license and development notes
  - FIXED: Fix Fuzz LaTeX test
  - FIXED: Fix bug in HR parsing
  - FIXED: Fix bug in determining label for headers
  - FIXED: Fix bug in printing NULL strings
  - FIXED: Fix bug in transclusion code
  - FIXED: Fix cmake to find on *nix systems libcurl
  - FIXED: Fix crash in parsing URL destinations
  - FIXED: Fix crash in reference definitions

- FIXED: Fix potential crash
- FIXED: Fix potential crash in definitions
- FIXED: Fix regression in transclusion
- FIXED: Free memory from asset hash
- FIXED: Improve Spanish translations
- FIXED: Improve token_split() algorithm and searching for abbreviations and glossary terms in text
- FIXED: Improve width/height attributes for HTML images
- FIXED: Update Spanish, update Spanish test suite, add languages to help screen
- FIXED: Update table caption alignment for CSS instead of deprecated alignment
- UPDATE: Update documentation
- UPDATED: Add notes to main.swift

- 2017–03–15 – v 6.0.0-rc1:

  - FIXED: Add missing CriticMarkup tokens to LaTeX
  - FIXED: Don't let labels end on " that is escaping the closing ']'
  - FIXED: Fix NULL pointer dereference
  - FIXED: Fix bug in Aho–Corasick implementation
  - FIXED: Fix bug with ATX Headers without newline
  - FIXED: Fix bug with Setext header starting with ':'
  - FIXED: Fix bug with leading spaces in abbreviation references
  - FIXED: Fix crash with empty definition
  - FIXED: Fix edge case with URL definitions
  - FIXED: Fix edge case with superscripts
  - FIXED: Fix null dereference error in CriticMarkup substitution
  - FIXED: Fix potential bug in Aho–Corasick search:
  - FIXED: Fix potential bug in storing items to hash
  - FIXED: Fix potential bug with line–>block parser
  - FIXED: Fix potential crash in attribute parsing
  - FIXED: Fix printing raw CriticMarkup tokens in LaTeX
  - FIXED: Fix signedness bug in Aho–Corasick
  - FIXED: Improve metadata edge cases; Fix NULL pointer dereference
  - FIXED: Include non–breaking space (ASCII 160) in re2c patterns

- FIXED: Keep ':' in false positive definitions
- FIXED: Lex space followed by tab as space, not text
- FIXED: Limit lines treated as ATX headers
- FIXED: Update test code

- 2017–03–13 – v 6.0.0-b2:

  - ADDED: Add CriticMarkup preprocessor that works across empty lines when accepting/rejecting markup
  - ADDED: Add back the mmd6 latex title file
  - ADDED: Basic EPUB 3 support – uses 'miniz' library to zip creation
  - ADDED: Update QuickStart and EPUB code
  - CHANGED: Update QuickStart guide
  - CHANGED: Update test suite
  - FIXED: Don't duplicate LaTeX glossary definitions
  - FIXED: Fix abbreviations in ODF; Improve test suite
  - FIXED: Improve glossaries and abbreviations; Update Quick-Start
  - FIXED: Tidy up some compiler warnings in code
  - FIXED: Use custom UUID code to minimize external dependencies

- 2017–03–09 – v 6.0.0-b1:

  - ADDED: Add French translations; fix typo in German
  - ADDED: Add Quick Start guide
  - ADDED: Add functionality to automatically identify abbreviations and glossary terms in source
  - ADDED: Improve LaTeX configuration files
  - ADDED: Update German translations
  - ADDED: Use native ODF table of contents instead of a manual list
  - ADDED: Use native command for table of contents in LaTeX
  - CHANGED: Bring HTML and ODF into line with LaTeX as to output of abbreviatinos on first and subsequent uses
  - CHANGED: Slight performance tweak
  - CHANGED: Update German test suite
  - FIXED: Allow {{TOC}} in latex verbatim

- – FIXED: Don't free token_pool if never initialized
- – FIXED: Fix German typo
- – FIXED: Fix missing token type
- – FIXED: Improve performance of checking document for metadata, which improves performance when checking for possible transclusion
- – FIXED: Update test suite for abbreviation changes

- 2017–03–05 – v 0.4.2-b:

  - – ADDED: Add and utility functions; fix memory leak
  - – ADDED: Initial abbreviation support
  - – ADDED: Keep working on Abbreviations/Glossaries
  - – ADDED: Refactor abbreviation code; Add inline abbreviations; Fix abbreviations in ODF
  - – ADDED: Update Inline Footnote test
  - – CHANGED: Add comments to i18n.h
  - – CHANGED: Finish refactoring note–related code
  - – CHANGED: Refactor footnotes
  - – CHANGED: Refactor glossary code
  - – CHANGED: Remove offset from html export functions
  - – FIXED: latex list items need to block optional argument to allow '[' as first character
  - – Merge branch 'release/0.4.1-b' into develop

- 2017–03–04 – v 0.4.1-b:

  - – FIXED: Add glossary localization

- 2017–03–04 – v 0.4.0-b:

  - – ADDED: Add TOC support to ODF
  - – ADDED: Add glossary support to ODF
  - – ADDED: Add prelim code for handling abbreviations
  - – ADDED: Add support for Swift Package Maker; CHANGED: Restructure source directory
  - – ADDED: Added LaTeX support for escaped characters, fenced code blocks, images, links
  - – ADDED: Basic ODF Support
  - – ADDED: Better document strong/emph algorithm

- ADDED: Continue ODF progress
- ADDED: Continue to work on ODF export
- ADDED: Continue work on ODF
- ADDED: Finish ODF support for lists
- ADDED: Improve performance when exporting
- ADDED: Improve token_pool memory handling
- ADDED: Prototype support for Glossaries
- ADDED: Support 'latexconfig' metadata
- CHANGED: Use multiple cases in glossary tests
- FIXED: Don't force glossary terms into lowercase
- FIXED: Fix Makefile for new source file location
- FIXED: Fix algorithm for creating TOC to properly handle 'incorrect' levels
- FIXED: Fix linebreaks in LaTeX; ADDED: Add Linebreaks test file
- FIXED: Fix new_source script for new directory structure
- FIXED: Fix non–breaking space in ODF
- FIXED: Fix padding at end of document body in ODF
- FIXED: Fix underscores in raw latex
- FIXED: Potential bug
- NOTE: Add shared library build option

- 2017–02–17 – v 0.3.1.a:

  - ADDED: 'finalize' beamer support
  - ADDED: Add escaped newline as linebreak; start on beamer/ memoir support
  - ADDED: CriticMarkup test for LaTeX
  - ADDED: Custom LaTeX output for CriticMarkup comments
  - ADDED: Support mmd export format
  - ADDED: Work on cpack installer – change project name for compatibility
  - CHANGED: Adjust latex metadata configuration for consistency
  - CHANGED: Configure cmake to use C99
  - FIXED: Add custom implementation for cross–platform support
  - FIXED: Fix German HTML tests

- FIXED: Fix cpack destination directory issue

- FIXED: Fix memory leaks etc

- FIXED: Fix warning in custom vasprintf

- FIXED: Modify CMakeLists.txt to test for use of clang compiler

- FIXED: Work on memory leaks

- NOTE: Adjust license width to improve display on smaller terminal windows

- 2017–02–14 – v 0.3.0a:

  - ADDED: Add basic image support to LaTeX

  - ADDED: Add file transclusion

  - ADDED: Add support for citation 'locators'

  - ADDED: Add support for manual labels on ATX Headers

  - ADDED: Add support for manual labels on Setext Headers

  - ADDED: Add support for tables in LaTeX

  - ADDED: HTML Comments appear as raw LaTeX

  - ADDED: Improved citation support in LaTeX

  - ADDED: Support \autoref{} in LaTeX

  - ADDED: Support combined options in LaTeX citations that use the '][' syntax

  - ADDED: Support language specifier in fenced code blocks

  - ADDED: Support metadata in LaTeX

  - ADDED: Update Citations test suite

  - FIXED: Escaped LaTeX characters

  - FIXED: Fix bug in URL parsing

  - FIXED: Fix bug in citation links

  - FIXED: Fix bug when no closing divider or newline at end of last table cell

  - FIXED: Fix issue printing '–'

  - FIXED: Fix scan_url test suite

  - FIXED: Get Math working in LaTeX

  - FIXED: Improve reliability or link scanner

  - FIXED: Properly add id attribute to new instances of citation only

  - FIXED: Properly handle manual labels with TOC

  - FIXED: Properly print hash characters in LaTeX

- – FIXED: Separate LaTeX verbatim and texttt character handling
  - – FIXED: Update Escapes test LaTeX result
  - – FIXED: Work on escaping LaTeX characters

- 2017–02–08 – v 0.1.4a:

  - – ADDED: Add smart quote support for other languages (resolves #15)

- 2017–02–08 – v 0.1.3a:

  - – ADDED: Add support for reference image id attributes
  - – ADDED: Add support for table captions
  - – ADDED: Metadata support for base header level
  - – ADDED: Support distinction between 3 and 5 backticks in fenced code blocks
  - – ADDED: Support Setext headers
  - – FIXED: Fix issue with metadata disrupting smart quotes

- 2017–02–07 – v 0.1.2a:

  - – "pathologic" test suite – fix handling of nested brackets, e.g. `[[[[foo]]]]` to avoid bogging down checking for reference links that don't exist.
  - – Table support – a single blank line separates sections of tables, so at least two blank lines are needed between adjacent tables.
  - – Definition list support
  - – "fuzz testing" – stress test the parser for unexpected failures
  - – Table of Contents support
  - – Improved compatibility mode parsing

- 2017–01–28 – v 0.1.1a includes a few updates:

  - – Metadata support
  - – Metadata variables support
  - – Extended ASCII range character checking
  - – Rudimentary language translations, including German
  - – Improved performance
  - – Additional testing:
    - * CriticMarkup
    - * HTML Blokcs
    - * Metadata/Variables
    - * "pathologic" test cases from CommonMark

Glossary

PEG Parsing Expression Grammar https://en.wikipedia.org/wiki/
Parsing_expression_grammar 1

Abbreviations

MD Markdown. 1

MMD MultiMarkdown. 1, 4–10